第十六节 服务架构升级

相关知识

Web 应用场景快速将请求对象交给底层服务（Service，如 Dubbo），

Web 应用接受 HTTP 请求，可能将请求封装为对象，调用下游服务。

Web 应用新生代空间可以设置得稍微大一点，大多数是短生命周期对象。尽可能不要缓存对象在 Web 应用 JVM 中。

回顾 Spring Web MVC

```
@RestController
public class DemoController{


    @RequestMapping("/save/user")
    public void saveUser(User user) { //
HandlerMethodArgumentResolver

    }

    @RequestMapping("/save/users")
```

```
    public void saveUsers(@NotNull @X List<User>
users) { // HandlerMethodArgumentResolver
        // 参数的类型 List.class             - Class
        // 参数的泛型参数类型 User - ParamertizedType
        // 更抽象的类型 Type - Class,
ParameterizedType
    }
}
```

JAX-RS V.S Spring Web

功能组件	JAX-RS	Spring Web
URI	UriBuilder	UriComponentsBuilder

RestTemplate

HttpMessageConverter 初始化

类似于 Spring Web MVC - org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport#addDefaultHttpMessageConverters

```
    protected final void
addDefaultHttpMessageConverters(List<HttpMessageCo
nverter<?>> messageConverters) {
        messageConverters.add(new
ByteArrayHttpMessageConverter());
        messageConverters.add(new
StringHttpMessageConverter());
        messageConverters.add(new
ResourceHttpMessageConverter());
        messageConverters.add(new
ResourceRegionHttpMessageConverter());
        if (!shouldIgnoreXml) {
            try {
                messageConverters.add(new
SourceHttpMessageConverter<>());
            }
            catch (Throwable ex) {
                // Ignore when no
TransformerFactory implementation is available...
            }
        }
        messageConverters.add(new
AllEncompassingFormHttpMessageConverter());

        if (romePresent) {
            messageConverters.add(new
AtomFeedHttpMessageConverter());
            messageConverters.add(new
RssChannelHttpMessageConverter());
        }

        if (!shouldIgnoreXml) {
            if (jackson2XmlPresent) {
```

```
                Jackson2ObjectMapperBuilder
builder = Jackson2ObjectMapperBuilder.xml();
                if (this.applicationContext !=
null) {

builder.applicationContext(this.applicationContext
);
                }
                messageConverters.add(new
MappingJackson2XmlHttpMessageConverter(builder.bui
ld()));
            }
            else if (jaxb2Present) {
                messageConverters.add(new
Jaxb2RootElementHttpMessageConverter());
            }
        }

        if (kotlinSerializationJsonPresent) {
            messageConverters.add(new
KotlinSerializationJsonHttpMessageConverter());
        }
        if (jackson2Present) {
            Jackson2ObjectMapperBuilder builder =
Jackson2ObjectMapperBuilder.json();
            if (this.applicationContext != null) {

builder.applicationContext(this.applicationContext
);
            }
            messageConverters.add(new
MappingJackson2HttpMessageConverter(builder.build(
)));
        }
        else if (gsonPresent) {
```

```
            messageConverters.add(new
GsonHttpMessageConverter());
        }
        else if (jsonbPresent) {
            messageConverters.add(new
JsonbHttpMessageConverter());
        }

        if (jackson2SmilePresent) {
            Jackson2ObjectMapperBuilder builder =
Jackson2ObjectMapperBuilder.smile();
            if (this.applicationContext != null) {

builder.applicationContext(this.applicationContext
);
            }
            messageConverters.add(new
MappingJackson2SmileHttpMessageConverter(builder.b
uild()));
        }
        if (jackson2CborPresent) {
            Jackson2ObjectMapperBuilder builder =
Jackson2ObjectMapperBuilder.cbor();
            if (this.applicationContext != null) {

builder.applicationContext(this.applicationContext
);
            }
            messageConverters.add(new
MappingJackson2CborHttpMessageConverter(builder.bu
ild()));
        }
    }
```

替换 HTTP Client 实现 - ClientHttpRequestFactory

提升需要性能

实现方法	API
基于 JDK HttpUrlConnection	org.springframework.http.client.SimpleClientHttpRequestFactory
基于 Http Components	org.springframework.http.client.HttpComponentsClientHttpRequestFactory
基于 OkHttp3	org.springframework.http.client.OkHttp3ClientHttpRequestFactory

WebFlux

基础设施

> The reactive-stack web framework, Spring WebFlux, was added later in version 5.0. It is fully non-blocking, supports Reactive Streams back pressure, and runs on such servers as Netty, Undertow, and Servlet 3.1+ containers.

Servlet 3.0 + - 异步特性

Servlet 3.1 + - 非阻塞

相关工程

Spring Boot 中 Netty-Reactor　Netty Web Server（代替 Servlet 引擎）

使用现状

WebFlux 用户数量严重不足，几乎大厂没有作为主流框架。WebFlux 与 Spring Cloud Gateway 不同的实现，两者尽管均为 Reactive Web 实现，底层均使用 Reactor 框架，不过设计的理念存在差异。两者的目标往 Reactive 变成模型上去适配。

编程模型

Reactive Streams 编程模型

Reactive 所有流程都是显示声明的，并且屏蔽同步和异步，还具备背压。

并发模型

异步

Spring Web MVC 异步特性

如果当前的 Servlet 引擎实现类似于 Tomcat 的话，那么异步操作会影响 Web 请求处理，Servlet 执行线程和异步执行线程都在 Tomcat HTTP 请求处理线程池上。

反之，Servlet 引擎 HTTP 线程池转发给外部线程池，同样面临 HTTP 请求方长时间等等。

假设 Servlet 线程池叫做 STP，外部线程池叫做 CTP，

如果 STP 线程数量（200）大于 CTP（100）的话，Web 客户端可能等待时间变长，吞吐量上去，响应时间。可能导致 Web 客户端出现等待超时。

反之，如果 STP 线程数量（200）小于 CTP（300）的话，Web 服务器响应时间下去，吞吐量也上去了，资源大量消耗，GC 以及 CPU 或 内存是否够用。假设，CTP 线程处理时 I/O 密集型的话，响应时间也可能下不去。举例说明，CTP 300 线程，DB 数据库连接池 50，250 线程在排队。如果 CPU 密集型的话，CPU 会线程上下文相对会比较频繁，Load 更高，甚至 GC 都没有资源安排。

Tomcat 默认 HTTP 处理线程池大小，core size： 10， max size：200

假设 200 个并发 HTTP 请求，访问资源同是异步 Servlet，

- T001 转发给 T101，T101 转发给 T001 -> 死锁
- T001 转发给 T101，T101 转发给 T002 -> 等待（可能长时间）

非阻塞

通常 Servlet 引擎中，很少直接与 Servlet 非阻塞特性或 API 打交道，往往 Servlet 引擎的底层实现已经切换成非阻塞模式，比如 Tomcat 7+：

Connector Comparison

Below is a small chart that shows how the connectors differ.

	Java Blocking Connector BIO	Java Non Blocking Connector NIO	APR/native Connector APR
Classname	Http11Protocol	Http11NioProtocol	Http11AprProtocol
Tomcat Version	since 3.x	since 6.0.x	since 5.5.x
Support Polling	NO	YES	YES
Polling Size	N/A	maxConnections	maxConnections
Read Request Headers	Blocking	Non Blocking	Blocking
Read Request Body	Blocking	Blocking	Blocking
Write Response	Blocking	Blocking	Blocking
Wait for next Request	Blocking	Non Blocking	Non Blocking
SSL Support	Java SSL	Java SSL	OpenSSL
SSL Handshake	Blocking	Non blocking	Blocking
Max Connections	maxConnections	maxConnections	maxConnections

Tomcat 从 6.0 开始实现 NIO（非阻塞）HTTP 协议连接器，然而它是实现 Servlet 2.5 规范。

Tomcat 7.0 开始实现 Servlet 3.0，而非阻塞 API 特性源于 Servlet 3.1+ 实现。

Servlet 3.1+ 非阻塞 API

HTTP 请求 - javax.servlet.ServletInputStream 中的 javax.servlet.ReadListener

HTTP 相应 - javax.servlet.ServletOutputStream 中的 javax.servlet.WriteListener

```
public interface ReadListener extends
EventListener {

    /**
     * When an instance of the
<code>ReadListener</code> is registered with a
{@link ServletInputStream},
     * this method will be invoked by the
container the first time when it is possible
     * to read data. Subsequently the container
will invoke this method if and only
     * if {@link
javax.servlet.ServletInputStream#isReady()} method
     * has been called and has returned
<code>false</code>.
     *
     * @throws IOException if an I/O related error
has occurred during processing
     */
    public void onDataAvailable() throws
IOException;

    /**
     * Invoked when all data for the current
request has been read.
     *
     * @throws IOException if an I/O related error
has occurred during processing
     */
```

```
    public void onAllDataRead() throws
IOException;

    /**
     * Invoked when an error occurs processing the
request.
     */
    public void onError(Throwable t);


}
```

参考文档:

http://tomcat.apache.org/tomcat-7.0-doc/aio.html

http://tomcat.apache.org/tomcat-7.0-doc/config/http.html#Connector_Comparison

函数编程接口 - RouterFunction

从特性来说，RouterFunction 类似于 Filter，依赖于 ServerRequest 和 ServerResponse。

```
public interface RouterFunction<T extends
ServerResponse> {

    /**
     * Return the {@linkplain HandlerFunction
handler function} that matches the given request.
     * @param request the request to route
     * @return an {@code Mono} describing the
{@code HandlerFunction} that matches this request,
     * or an empty {@code Mono} if there is no
match
     */
    Mono<HandlerFunction<T>> route(ServerRequest
request);
}
```

ServerRequest 类似于 HttpServletRequest

ServerResponse 类似于 HttpServletResponse

在 Servlet 中，Filter API 通过 doFilter 方法传递 ServletRequest 和 ServletResponse 对象 以及 FilterChain。其中，FilterChain 是控制 Filter 是否要继续到下一个 Filter 的开关。

在 RouterFunction 中，route 方法来返回 Mono<HandlerFunction>：

- 当 Mono 对象包含 HandlerFunction 时，说明有处理动作
- 否则，就没有

Q1：是否可以认为 RouterFunction 类似于网关或路由?

A：类似，HandlerFunction 类似于 Servlet API，从语义上而言，RouterFunction 通过一定路由规则转发到某个 HandlerFunction。

Q2：RouterFunction 是如何实现类似于 Filter 责任链的?

RouterFunction 提供了 and 等默认方法，它可以按照需要去组合其他 RouterFunction。

- and 或 andOther - 组合其他 RouterFunction
- andRoute - 增加新的路由规则和 HandlerFunction